

Laboratory work 4: **Operational** **Systems**

Elaborated:
st. gr. FAF-213

Konjevic Alexandra

Verified:

Rastislav Călin

Topic: Boot Loader

Objectives: Create in assembly language an application that will act as a Boot Loader and will perform the following tasks:

1. It will display a greeting message that includes the author's name and will await keyboard input for the 'source' address on the floppy disk, from where the kernel (or other compiled code intended to be loaded and executed) will be read. The address should be entered in the SIDE, TRACK, SECTOR format and must strictly fall within the range reserved for the author student, as it was in Lab3.
2. It will wait for keyboard input for the 'destination' address in RAM where the data block read from the floppy disk will be loaded. The RAM address should be in the format XXXXh:XXXXh, identical to how it was in Lab3.
3. It will transfer the FLOPPY ==> RAM data and display the error code with which the operation was completed.
4. It will display a message to press a key to launch the kernel (or execute the desired code).
5. Upon completion of the kernel execution or executed code, it will display a message to press a key to repeatedly execute the Boot Loader.

Implementation:

To create a boot loader system capable of loading a kernel or other compiled code from the floppy disk into RAM, and then, executing it, there were created two bootloaders: `'bootloader.asm'` and `'mini_boot.asm'`. The first part (`'bootloader.asm'`) contains the main functionality, while the second part (`'mini_boot.asm'`) is a small boot loader meant to load the main boot loader.

The code begins with the `org '7d00h'` directive, setting the origin point where the bootloader code will reside in memory.

Various data sections (`'section .data'` and `'section .bss'`) are declared to hold prompts, user inputs, and intermediate variables:

```

1  section .data:
2      prompt db 'Welcome, Alexandra. Press any key to continue: $'
3      hts_prompt db "Enter CHTS script address$"
4      so_prompt db "Enter XXXX:YYYY RAM address$"
5      kernel_start db "Press any key to load the kernel: $"
6
7      sc db 0
8      h db 0
9      t db 0
10     s db 0
11
12     c db 0
13     result db 0
14
15     marker db 0
16
17
18  section .bss:
19     hex_result resb 2
20     add1 resb 2
21     add2 resb 2
22     buffer resb 2

```

The line that follows after this - ``dw 0AA55h`` - is marking the end of a boot sector, indicating to the BIOS that the sector is bootable and the code contained within it should be executed during the boot process.

The next step is displaying the initial prompt (*Welcome, Alexandra. Press any key to continue: \$*):

```

1  ; print initial prompt
2  mov si, prompt
3  call print

```

The function ``print`` is being called:

```

1  print:
2      call cursor
3
4      print_char:
5          mov al, [si]
6          cmp al, '$'
7          je end_print
8
9          mov ah, 0eh
10         int 10h
11         inc si
12         jmp print_char
13
14     end_print:
15         ret

```

This function also calls another function: ``cursor`` which sets the position of the cursor using the function ``03h`` of the interruption ``10h``:



```
1  cursor:
2      mov ah, 03h
3      mov bh, 0
4      int 10h
5
6      ret
7
```

Next, the printing function declares the ``print_char`` label, which loads the character from the memory address pointed to by ``si`` into the ``al`` register and compares the character loaded with the ASCII value of ``$``. This symbol ``$`` marks the end of the prompt. If the character loaded is ``$``, it jumps to the ``end_print`` label, which effectively ends the printing process.

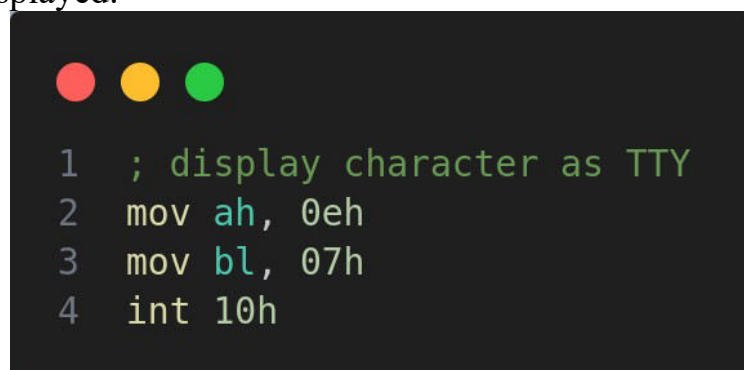
Printing characters occurs in the following mode: the function sets up the video display function (``0eh``) of BIOS interrupt ``10h`` for displaying characters. After that, there is incremented the memory address in ``si`` to point to the next character in the string and the function jumps back to the ``print_char`` label to continue printing the next character.

After the prompts are displayed, the user input is read, using the function ``00h`` of the interruption ``16h``:




```
1  ; read option
2  mov ah, 00h
3  int 16h
```

And this character is displayed:



```
1  ; display character as TTY
2  mov ah, 0eh
3  mov bl, 07h
4  int 10h
```

After that, there is printed a new line, by calling the function ``newline``:




```

1  newline:
2      call cursor
3
4      mov ah, 02h
5      mov bh, 0
6      inc dh
7      mov dl, 0
8      int 10h
9
10     ret

```

This function sets the position of the cursor on a new line.

Next, the next prompt is displayed, to get the CHTS script address (*Enter CHTS script address\$*):




```

1  mov si, hts_prompt
2  call print
3  call newline

```

The next block of code is responsible for displaying the `>` symbol in front of the user input:



```

1  ; print sector count prompt
2  mov ah, 0eh
3  mov al, '>'
4  mov bl, 07h
5  int 10h


```

Next, we have the code:



```
1  mov byte [result], 0
2  call clear
3  call read_buffer
4
5  mov al, [result]
6  mov byte [sc], al
7
8  call newline
```

This instruction moves the value 0 into the memory location pointed to by the label `[result]`. After that, the functions `clear` and `read_buffer` are called:



```
1  clear:
2      mov byte [c], 0
3      mov byte [si], 0
4      mov si, buffer
5
6      ret
```

The `clear` function sets the value 0 to the memory locations pointed by `c` and `si`, and moves the value of the `buffer` to the register `si`.

Next, we have the `read_buffer` function:

```

1  read_buffer:
2
3      read_char:
4          ; read character
5          mov ah, 00h
6          int 16h
7
8          ; check if the ENTER key was introduced
9          cmp al, 0dh
10         je hdl_enter
11
12         ; check if the BACKSPACE key was introduced
13         cmp al, 08h
14         je hdl_backspace
15
16         ; add character into the buffer and increment its pointer
17         mov [si], al
18         inc si
19         inc byte [c]
20
21         ; display character as TTY
22         mov ah, 0eh
23         mov bl, 07h
24         int 10h
25
26         jmp read_char
27
28     hdl_enter:
29         mov byte [si], 0
30         mov si, buffer
31
32         cmp byte [marker], 0
33         je atoi_jump
34         jmp atoh_jump
35
36     hdl_backspace:
37         call cursor
38
39         cmp byte [c], 0
40         je read_char
41
42         ; clear last buffer char
43         dec si
44         dec byte [c]
45
46         ; move cursor to the left
47         mov ah, 02h
48         mov bh, 0
49         dec dl
50         int 10h
51
52         ; print space instead of the cleared char
53         mov ah, 0ah
54         mov al, ' '
55         mov bh, 0
56         mov cx, 1
57         int 10h
58
59         jmp read_char
60
61     atoi_jump:
62         call atoi
63         jmp end_read_buffer
64
65     atoh_jump:
66         call atoh
67         jmp end_read_buffer
68
69     end_read_buffer:
70
71     ret

```

First of all, it read a character from the keyboard (with function `00h` of the `16h` interruption), after that, handles the enter or backspace keys. If the keys are neither of them, then the introduced character is added to the buffer, and the buffer pointer is incremented. The character read from the keyboard is also displayed on the screen.

The handling of the enter key uses also additional helper functions:

```

1  atoi:
2      xor ax, ax
3      xor bx, bx
4
5      atoi_d:
6          lodsb
7
8          sub al, '0'
9          xor bh, bh
10         imul bx, 10
11         add bl, al
12         mov [result], bl
13
14         dec byte [c]
15         cmp byte [c], 0
16         jne atoi_d
17
18     ret

```

This function is responsible for converting a sequence of ASCII characters representing a decimal number into its corresponding numerical value. The instructions `xor ax, ax` and `xor bx, bx` set the values of these registers to zero. `atoi_d` label marks the beginning of the loop where the actual conversion takes place. `lodsb`: This instruction loads the byte addressed by the `si` register into the `al` register and increments `si` automatically, this means it loads the next character from memory into `al` on each iteration.

Conversion steps: `sub al, '0'` subtracts the ASCII value of character '0' from the ASCII value of the current character in `al`. ASCII values are numeric representations of characters, and subtracting the ASCII value of '0' effectively converts the character to its numerical value. Next, the `bh` register is cleared. `imul bx, 10` multiplies the value in `bx` (which initially holds the previously processed part of the number) by 10. This prepares for the next digit to be added. `imul` stands for "integer multiply". It effectively multiplies BX by 10 and stores the result back in `bx`. `add bl, al` adds the converted numerical value in `al` to the `bl`. It accumulates the digits to form the final integer value. Next, this value is stored into the memory location labeled as `result`. This is the accumulating result of the conversion. `dec byte [c]` decrements the byte in memory pointed to by `c`. `cmp byte [c], 0`: Compares the value pointed to by `c` with zero to check if the conversion is complete. If there are more characters to convert (`[c]` is not zero), the loop continues (`jne atoi_d`) to process the next character. If all characters have been processed, the subroutine proceeds to `ret`, indicating the end of the conversion.

Also, the handling of enter uses the function:


```

1  atoi:
2      xor bx, bx
3      mov di, hex_result
4
5      atoi_s:
6          xor ax, ax
7          mov al, [si]
8
9          cmp al, 65
10         jg atoi_l
11         sub al, 48
12         jmp continue
13
14         atoi_l:
15             sub al, 55
16             jmp continue
17
18         continue:
19             mov bx, [di]
20             imul bx, 16
21             add bx, ax
22             mov [di], bx
23
24             inc si
25
26             dec byte [c]
27             jnz atoi_s
28
29         ret

```

The `atoi` (ASCII to Hexadecimal) subroutine is responsible for converting a sequence of ASCII characters representing a hexadecimal number into its corresponding numerical value.

First of all, it clears the value of register `bx` and sets the value of register `di` to the value of `hex_result`. `atoi_s` label marks the beginning of the loop where the actual conversion takes place.

`mov al, [si]` loads the byte addressed by the `si` register into the `al` register. This retrieves the next character from memory for processing. After that, if the value from `al` is greater than 65, the code jumps to `atoi_l`. If the character is less than or equal to '9', it proceeds directly to the next step. `jg atoi_l` (if the character is greater than 'A', it means the character is in the range 'A' to 'F').

The function subtracts 55 from the ASCII value (`al`), converting characters 'A' to 'F' into their respective hexadecimal values ('A' = 10, 'B' = 11, ..., 'F' = 15). `sub al, 48`: If the character is '0' to '9', subtracts 48 from the ASCII value (`al`) - this converts ASCII characters '0' to '9' into their respective numerical values (0 to 9).

``continue`` label moves the current value in ``bx`` into memory location ``[di]`` (`hex_result`); multiplies the value in ``bx`` by 16 to shift it left by one hexadecimal digit. After that, it adds the converted value in ``ax`` to ``bx`` and stores the updated value in ``[di]``. It then increments the ``si`` register to move to the next character and decrements the byte in memory pointed to by ``c`` to track the length of the input string. If there are more characters to convert (``[ci]`` is not zero), the loop continues (``jnz atoh_s``).

The handling of the backspace is done this way:

```
1  hdl_backspace:
2      call cursor
3
4      cmp byte [c], 0
5      je read_char
6
7      ; clear last buffer char
8      dec si
9      dec byte [c]
10
11     ; move cursor to the left
12     mov ah, 02h
13     mov bh, 0
14     dec dl
15     int 10h
16
17     ; print space instead of the cleared char
18     mov ah, 0ah
19     mov al, ' '
20     mov bh, 0
21     mov cx, 1
22     int 10h
23
24     jmp read_char
```

The ``hdl_backspace`` segment checks if there's a character in the buffer to be deleted. If there's a character present, it erases the last character in the buffer, moves the cursor back one position on the screen, prints a space character to visually erase the character on the display, and then resumes waiting for new input by jumping back to the read character section. If there are no characters in the buffer, it continues to wait for new input without any erasing action.

So, there are read from the input the values of the sector count:

```
1  ; print sector count prompt
2  mov ah, 0eh
3  mov al, '>'
4  mov bl, 07h
5  int 10h
6
7  mov byte [result], 0
8  call clear
9  call read_buffer
10
11  mov al, [result]
12  mov byte [sc], al
13
14  call newline
```

And in the same way there are read the head, track and sector values. After that, also the values of

offset and segment, and after that, when we have all the values, the function `load_kernel` is called:

```
1  load_kernel:
2      mov ah, 0h
3      int 13h
4
5      mov ax, [add2]
6      mov es, ax
7      mov bx, [add1]
8
9      ; load the NASM script into memory
10     mov ah, 02h
11     mov al, [sc]
12     mov ch, [t]
13     mov cl, [s]
14     mov dh, [h]
15     mov dl, 0
16
17     int 13h
18
19     ; print error code
20     mov al, '0'
21     add al, ah
22     mov ah, 0eh
23     int 10h
24
25     call newline
26
27     ret
```

Individual task:

My individual task consists in counting the number of occurrences of a character in a string. Both the string and character are given from input.

First of all, the program receives the segment and offset values from bootloader. They are stored in the variables `add1` and `add2`. Also, here are initialized other variables of the program:

```
1  _start:
2      ; receive segment:offset pair from the bootloader
3      mov [add1], ax
4      mov [add2], bx
5
6      mov byte [string], 0
7
8      mov word [character], 0
9
10     mov byte [char_counter], 0
11     mov byte [result], 0
12
13     mov byte [c], 0
14
15     jmp menu
```

After that, there is the menu of the program. Initially, I set there the video mode, clear the screen and find the position of the cursor, using the function `find_current_cursor_position`:

```

1  find_current_cursor_position:
2      push ax
3      push bx
4      push cx
5      mov ah, 03h
6      mov bh, byte 0
7      int 10h
8      pop cx
9      pop bx
10     pop ax
11     ret

```

This function resets the registers `ax`, `bx` and `cx` and then, using the function `03h` of the interruption `10h` (function to query cursor position and size), obtains the current cursor position and the size/shape of the cursor for a specified video page, storing the row and column values in the `dh` and `dl` registers.

After that, the program prints the disclaimer. For every string predefined in the data section of the program, I add to the register that points to them the value of the offset, stored in the `add1`.

```

1  mov bh, 0
2      mov bl, 07h
3
4      mov si, disclaimer
5      add si, word [add1] ; add offset to si
6      call print_string_inline ; prints the string pointed by si
7
8      call newline

```

Here, I use two helper functions: `newline` and `print_string_inline`:

```

1  newline:
2      call find_current_cursor_position ; dh = row, dl = column
3
4      mov ah, 02h
5      mov bh, 0
6      inc dh ; on the next row compared to the current one
7      mov dl, 0 ; at the beginning of the row
8      int 10h
9
10     ret

```

Newline executes the `find_current_cursor_position` and stores the values of the cursor in `dh` and `dl`, and increments the value of the row, where the cursor should be placed, and sets the value of the column to zero, thus, the cursor is on the beginning of the next line.

```

1  ;; Print string at cursor position and move cursor at the end of it
2  ;; Parameters: bh - page number
3  ;;             bl - video attribute http://www.techhelpmanual.com/87-screen\_attributes.html
4  ;;             si - pointer to string
5  ;; Returns: None
6  print_string_inline:
7      pusha
8      mov bh, 0
9      ;; Get cursor position
10     mov ah, 03h
11     int 10h
12     ;; Get string length
13     call str_len ; cx = string length
14     mov ax, 1301h
15     mov bp, si
16     int 10h
17     popa
18     ret

```

The helper function above expects a pointer to the string in the ``si`` register, that it will print. To print the string, we need to have its length, for which I use another helper function - ``str_len`` that returns the length in the ``cx`` register. This is the function ``str_len``:

```
1  ;; Gets string length
2  ;; Parameters: si - pointer to string
3  ;; Returns:    cx - string length
4  ;; Notes:      String must be zero terminated
5  str_len:
6      mov cx, 0
7      mov [pointer_store], si
8      cmp byte [si], 0
9      je .str_len_end
10
11     ; increment cx and si until the null terminator is reached
12     .str_len_loop:
13         inc cx
14         inc si
15         cmp byte [si], 0
16         jne .str_len_loop
17
18     .str_len_end:
19         mov si, [pointer_store] ; restore si
20         ret
21
```

It expects a null terminated string that is pointed to by ``si`` register. The function loops through every character of the string, until it reaches the null terminator. The length is returned in the ``cx`` register. Next, in the initial menu, we have the function to reboot the window and return to the initial screen, by pressing the ``r`` key. If it is not pressed, the program proceeds to reading from input the values of the string and character that needs to be find in the string. For reading the input, I use the functions ``read_buffer``, that adds characters in the buffer, and as soon as the enter key is pressed, the value from the buffer is first stored in the ``string`` and after that, in ``character``, and the function ``clear_buffer``, that resets the buffer and the character counter.

After the value of string and character are read from input, the prompt for the result is displayed on the screen, and also, the ``find_character_occurrence`` function is executed. First of all, it initializes the counter for the occurrences in the string; it sets the beginning of the string in the register ``si``; sets the character to find in the register ``al``; sets counter to 0.

```
1  ; initialize counter
2  mov byte [counter], 0
3
4  ; find character occurrence
5  mov si, string ; set si to the beginning of the string
6  mov al, byte [character] ; character to find
7  mov cx, 0 ; occurrences counter
```

After that, there is the loop to find occurrences. It advances through every character of the string, and compares it with the value pointed by ``al``. If there is an occurrence, the ``cx`` value is incremented, and if not, the pointer moves to the next character of the string, and executes the loop again:

```
1  find_occurrence_loop:
2      cmp byte [si], 0 ; if the string is empty
3      je print_occurrence
4      cmp byte [si], al ; if the character is found
5      je found_occurrence
6      jne loop_end
7
8      found_occurrence:
9          inc cx ; increment occurrences counter
10
11     loop_end:
12         inc si ; move to the next character
13         jmp find_occurrence_loop ; continue looping for the next character
14
```

When the loop ends, the number of occurrences is printed:

```
1 print_occurrence:
2     ; Display the occurrences
3
4     cmp cx, 0
5     je display_not_found
6
7     ; Convert count (BL register) to string
8     pusha
9     call find_current_cursor_position
10    mov ax, cx ; move count to ax
11
12    mov di, another_buffer
13    call int_to_string ; converts string pointed by di (another_buffer) to string
14
15    mov bh, 0 ; Page number
16    mov bl, 07h ; Text attribute
17    mov si, another_buffer
18    call print_string_inline
19
20    popa
21
22    jmp end_find_character_occurrence
```

Another helper function is used here: a function to convert the integer value of occurrences in a string value, for it to be displayed on the screen:

```
1 ;; Converts uint to string
2 ;; Parameters: ax - uint to convert
3 ;;             di - buffer to store string
4 ;; Returns:     Nothing
5 ;; Mutates:     di
6 int_to_string:
7     pusha
8     mov bx, 10 ; Move constant 10 to bx for division
9     mov cx, 0 ; Initialize counter for the number of digits
10    .int_to_string_loop:
11        xor dx, dx ; Clear dx for division
12        div bx ; Divide ax by 10, quotient in ax, remainder in dx
13        push dx ; Push remainder (digit) onto stack
14        inc cx
15        cmp ax, 0
16        jne .int_to_string_loop ; If not zero, continue looping
17    .int_to_string_loop2:
18        pop dx ; Pop digit from stack to dx
19        add dl, '0' ; Convert digit to ASCII character
20        mov [di], dl ; Store ASCII character in buffer
21        inc di ; Move to the next position in the buffer
22        loop .int_to_string_loop2 ; Continue until all digits are processed
23    mov byte [di], 0 ; Null-terminate the string in the buffer
24    popa
25    ret
```

This function mutates the value pointed by the 'di' register, and converts the value from an integer to a string. When the number of occurrences is printed, the function waits for any key to be pressed, to load the menu again.

Running the code

In order to create a bootable image for the virtual machine, first of all we need to assemble the program from the file with the individual task, using the command `nasm task.asm`. This command creates a binary file. After that, we need to execute the script `boot_floppy_script.sh`, passing it three parameters – the binary file, the bootloader (`bootloader.asm`) and the second bootloader (`mini_boot.asm`): `./boot_floppy_script.sh rainbow bootloader.asm mini_boot.asm`. The script is the following:

```
1 #!/bin/bash
2
3 binary_file="$1"
4 bootloader="$2"
5 mini_bootloader="$3"
6
7 # Check if the binary file exists
8 if [ ! -f "$binary_file" ]; then
9     echo "Error: Binary file '$binary_file' does not exist."
10    exit 1
11 fi
12
13 nasm -f bin $mini_bootloader -o mini_bootloader.bin
14 nasm -f bin $bootloader -o bootloader.bin
15
16 # Create an empty floppy disk image (1.44MB size)
17 floppy_image="floppy.img"
18 truncate -s 1474560 mini_bootloader.bin
19 mv mini_bootloader.bin $floppy_image
20
21 dd if="bootloader.bin" of="$floppy_image" bs=512 seek=1 conv=notrunc
22 dd if="$binary_file" of="$floppy_image" bs=512 seek=3 conv=notrunc
23
24 echo "Binary file '$binary_file' successfully added to floppy image '$floppy_image'."
```

The script takes three arguments from the command line: \$1 (binary_file), \$2 (bootloader), and \$3 (mini_bootloader). It checks if the binary file provided as \$1 exists. If it doesn't exist, it displays an error message and exits the script with a non-zero status. It uses NASM with the `-f bin` option to assemble the `mini_bootloader` file into `mini_bootloader.bin`. Similarly, it assembles the `bootloader` file into `bootloader.bin`.

After that, the script creates an empty floppy disk image named `floppy.img` with a size of 1.44MB using the truncate command. It then is essentially moving the `mini_bootloader.bin` file and renaming it to the name stored in the variable `$floppy_image` (which is `floppy.img` as per the script).

Using dd (data duplicator), it writes the `bootloader.bin` content to the created floppy image starting at the second block (512 bytes per block) using `seek=1`. This places the bootloader in the boot sector. Similarly, it writes the content of the binary file (`$binary_file`) to the floppy image starting at the fourth block (`seek=3`). This presumably adds the program or kernel to the floppy image.

Finally, it prints a message indicating that the binary file provided (`$binary_file`) has been successfully added to the floppy image (`$floppy_image`).

CONCLUSIONS

During this laboratory work, we worked together to create a flexible assembly program with different functions. The program starts with a menu where users can choose operations involving the keyboard, floppy disk, or RAM. It asks for various inputs like the number of operations, head, track, sector, data, segment, offset, and memory address.

The program handles user input effectively and performs tasks like writing from the keyboard to the floppy disk, reading from the floppy disk to RAM, and writing from RAM to the floppy disk. It shows error messages and can repeat operations.

For hexadecimal input, the program converts it into numbers and adjusts its behavior based on user choices. Users can easily move through different prompts, and the display updates accordingly.

Repository: <https://github.com/alya1007/Labs-semester-5/tree/master/OS/lab4>